C introduction

Complex data types

Richard Mörbitz, Manuel Thieme

Contents

Introduction

Composite data types

Enumerations

Style

Exercise

Enumerations

Fxercise

Motivation

Think of a data type that can store all data belonging to a person:

```
char name[32];
int age, id;
```

However, there seems to be no way to put those different types together.

Think of a data type that can store the state (current color) of traffic lights:

```
int color:
/* 0 = red, 1 = yellow, 2 = green */
```

How to avoid someone assigning a different value to color?

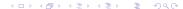


Primitive data types are fine as long as you want to

- ▶ Store a single value that does not depend on other variables
- ▶ Store a sequence of values of the same type with a constant length
 - \rightarrow arrays

However, it is not possible to

- ► Compose variables of different data types to a compound structure
 - \rightarrow composite data types
- ▶ Have a variable that can only attain certain values
 - \rightarrow enumerations
- Have a sequence with an adjustable length
 - \rightarrow soon...



Introduction

0

Data records

Composite data types are derived from primitive data types. You can store any number of primitive variables in one composite variable.

- ▶ The composite variable is called *structure* and has the type *struct*
- ▶ The primitive variables are called *members* of that structure

Defining a new composite type "struct person":

```
struct person {    /* struct <identifier > */
    int id;
    int age;    /* block for member declaration */
    char name[32];
};    /* end declaration with ';' */
```

A struct variable is at least as large as all of its members.



struct variables

Our new type *struct person* can be used to declare variables any where in its scope:

```
struct person pers_alice , pers_bob;
```

You can declare a *struct* variable directly in the type definition:

```
struct person {
    /* member declaration */
} pers_alice , pers_bob;
```

If we do not need the struct type *person* for further variable declarations, its identifier can be left out.



To initialize the *struct* members upon declaration, enclose the values in

braces as we did it for arrays:

```
struct person pers_alice = { 1, 20, "Alice" };
```

To access the struct members, use the struct identifier followed by a '.' and the member identifier:

```
printf("%d\n", pers_alice.id);
pers_alice.age++;
```

An adress is rather complicated:

```
struct adress {
   int postcode;
   /* ... imagine much more members */
};
```

Now, let the *person* have one:

```
struct person {
    struct adress contact;
    /* ... and all the other members */
} pers_alice;
```

Access:

```
pers\_alice.contact.postcode = 15430;
```



- ▶ Similar to *structs*, handle them in the same way
- However: only one member can be "active"
- ▶ If you assign a value to a member, all other members become invalid

Interface between a *list* and a *vector* implementation:

```
union compound {
    int list [3];
    struct {
        int x1, x2, x3;
    } vector;
};
```

The size of a union variable is equal to the size of its largest member.

ightarrow saving memory



An enumeration consists of identifiers that behave like constant values. It is declared using the keyword *enum*:

```
enum light {
    RED.
    YELLOW.
    GRFFN
};
```

Now you can assign the values red, yellow and green to variables of the type enum light. Internally they are represented as numbers (red = 0, yellow = 1 etc.), but

- Using the aliases is clear and fancy
- ▶ No invalid values (like -1) can be assigned



Profit

You can determine the values of the constants on your own:

However, this can confuse people \rightarrow only use it if there is a good reason.

Enumerations provide a nice way to define "global" constants:

```
enum { WIDTH = 10, HEIGHT = 20 };
...
char tetris_board[WIDTH][HEIGHT];
```



Consistency

- ► Since complex type definitions heavily rely on blocks, you should use the same coding conventions on them
- Let your custom type identifiers start with small letters

If you define a complex data type, you are very likely going to use it in many different parts of your program.

ightarrow Have a global type definition, declare the variables in the local context

Name *enum* constants in CAPITAL letters to visually seperate them from variables.



typedef

Sometimes you see people writing code like that:

```
typedef struct foo {
   /* member declarations */
} bar;
```

This creates the new type bar which is nothing more than a struct foo.

However, this simple fact is hidden for other programmers working on the same project \rightarrow **possible confusion**.

- Unclear, if bar is a composite type at all
- ▶ If so, is it a *struct* / *union* / *enum* or something really crazy?



Introduction

Please, avoid using typedef.1

Never use typedef.

Introduction

Please, avoid using typedef.1

Of course, there are situations in which the use of typedef makes sense. BUT:

- ▶ Not in the C introduction course
- ► Not for simple *structs*



¹Seriously, never use *typedef*.

Complex numbers

A complex number a + bi can be handled as a composite of two real numbers (real part a and imaginary part b).

- Write a program that is able to read and print complex numbers and stores them in an appropriate data structure. The output should be in the form a + bi, but you can ask to input a and b seperately.
- **Experts:** write a function that takes a *complex number* as an argument and returns its absolute value.



A complex number a + bi can be handled as a composite of two real numbers (real part a and imaginary part b).

Write a program that is able to read and print complex numbers and stores them in an appropriate data structure.

Enumerations

- The output should be in the form a + bi, but you can ask to input a and b seperately.
- **Experts:** write a function that takes a *complex number* as an argument and returns its absolute value.
 - ▶ Hint: You have to pass a *complex number*, but the return value is a real number



Circles

A point in 2D consists of two coordinates: x and y (both float). A circle consists of a centre (point), a radius, a circumference and an area (all float).

- Write a program that reads two coordinates and a radius from the command line and stores them in the *struct circle* described above. Then the circumference and area are calculated and stored as well. Afterwards the updated *struct circle* is printed.
- **Experts:** Write a function that takes two *circles* as arguments and returns a *circle* that is goes through their *centres*.